A Beginner’s Guide to MATLAB*

Christos Xenophontos
Department of Mathematics & Statistics
University of Cyprus

*MATLAB is a registered trademark of The MathWorks Inc. A first version of this document appeared as Technical Report 98-02, Department of Mathematics & Computer Science, Clarkson University, NY, USA.
# TABLE OF CONTENTS

1. **Introduction**
   1.1 Accessing MATLAB  3
   1.2 How to read this tutorial  4

2. **MATLAB Basics**
   2.1 The basic features  5
   2.2 Vectors and matrices  7
   2.3 Built-in functions  13
   2.4 Plotting  21

3. **Programming in MATLAB**
   3.1 M-files: Scripts and functions  26
   3.2 Loops  27
   3.3 If statement  31

4. **Additional Topics**
   4.1 Polynomials in MATLAB  36
   4.2 Numerical Methods  37
   4.3 Numerical solution of ODEs  41

5. **Closing Remarks and References**  44
1. INTRODUCTION

MATLAB, which stands for MATrix LABoratory, is a state-of-the-art mathematical software package, which is used extensively in both academia and industry. It is an interactive program for numerical computation and data visualization, which along with its programming capabilities provides a very useful tool for almost all areas of science and engineering. Unlike other mathematical packages, such as MAPLE or MATHEMATICA, MATLAB cannot perform symbolic manipulations without the use of additional Toolboxes. It remains however, one of the leading software packages for numerical computation. As you might guess from its name, MATLAB deals mainly with matrices. A scalar is a 1-by-1 matrix and a row vector of length say 5, is a 1-by-5 matrix. We will elaborate more on these and other features of MATLAB in the sections that follow. One of the many advantages of MATLAB is the natural notation used: it looks a lot like the notation that you encounter in a linear algebra course. This makes the use of the program especially easy and it is what makes MATLAB a natural choice for numerical computations.

The purpose of this tutorial is to familiarize the beginner to MATLAB, by introducing the basic features and commands of the program. It is in no way a complete reference and the reader is encouraged to further enhance his or her knowledge of MATLAB by reading some of the suggested references at the end of this guide. Moreover, very little attempt has been made to present the optimal way of coding, as this is a beginner’s guide.

1.1 Accessing MATLAB

MATLAB is available at (most) universities laboratory PCs. If you are an undergraduate student, currently enrolled in a university, you may purchase MATLAB from the company’s webpage (www.mathworks.com) at a very reasonable price.

To start MATLAB on a PC, you double-click the MATLAB icon (assuming there is one on the Desktop)

and then a window will open, much like the one in Figure 1 below. The Command Window, as it is called, is located in the middle and that is where we will type the commands, as described in the next subsection.
1.2 How to read this tutorial

In the sections that follow, the MATLAB prompt (>>) will be used to indicate where the commands are entered. Anything you see after this prompt denotes user input (e.g. a command) followed by a carriage return (i.e. the “enter” key). We type the commands in the Command Window. Often, input is followed by output so unless otherwise specified, the line(s) that follow a command will denote output (i.e. MATLAB’s response to what you typed in). MATLAB is case-sensitive, which means that a + B is not the same as a + b. Different fonts, like the ones you just witnessed, will also be used to simulate the interactive session. This can be seen in the example below.

MATLAB can work as a calculator. If we ask MATLAB to add two numbers, we get the answer we expect.

```matlab
>> 3 + 4
ans =
    7
```

As we will see, MATLAB is much more than a “fancy” calculator. In order to get the most out this tutorial, you are strongly encouraged to try all the commands introduced in each section and work on all the recommended exercises. This usually works best if after reading this guide once, you read it again (and possibly again and again) in front of a computer.
2. MATLAB BASICS

2.1 The basic features

Let us start with something simple, like defining a row vector with components the numbers 1, 2, 3, 4, 5 and assigning it a variable name, say $x$. In MATLAB, we type

```matlab
>> x = [1 2 3 4 5]
```

and we get

```
x =
1     2     3     4     5
```

Note that we used the equal sign for assigning the variable name $x$ to the vector, brackets to enclose its entries and spaces to separate them. (Just like you would do using the usual linear algebra notation). We could have used commas (, ) instead of spaces to separate the entries, or even a combination of the two. The use of either spaces or commas is necessary!

To create a column vector (MATLAB distinguishes between row and column vectors, as it should) we can either use semicolons (;) to separate the entries, or first define a row vector and take its transpose to obtain a column vector. Let us demonstrate this by defining a column vector $y$ with entries 6, 7, 8, 9, 10 using both techniques.

```matlab
>> y = [6;7;8;9;10]
y =
6
7
8
9
10
```

```matlab
>> y = [6,7,8,9,10]
y =
6     7     8     9    10
```

```matlab
>> y'
ans =
6
7
8
9
10
```
Let us make a few comments. First, note that to take the transpose of a vector (or a matrix for that matter) we use the single quote ('). Also note that MATLAB repeats (after it processes) what we typed in. Sometimes, however, we might not wish to “see” the output of a specific command. We can suppress the output by using a semicolon (;) at the end of the command line. Finally, keep in mind that MATLAB automatically assigns the variable name ans to anything that has not been assigned a name. In the example above, this means that a new variable has been created with the column vector entries as its value. The variable ans, however, gets recycled and every time we type in a command without assigning a variable, ans gets that value. It is good practice to keep track of what variables are defined and occupy our workspace. Due to the fact that this can be cumbersome, MATLAB can do it for us. The command whos gives all sorts of information on what variables are active.

```
>> whos
Name      Size            Bytes  Class     Attributes
x         1x5                40  double
y         5x1                40  double
```

A similar command, called who, only provides the names of the variables that are active.

```
>> who
Your variables are:
ans x y
```

If we no longer need a particular variable, we can “erase” it from memory using the command clear variable_name. Let us clear the variable ans and check that we indeed did so.

```
>> clear ans
>> who
Your variables are:
x y
```

The command clear used by itself, “erases” all the variables from the memory. Be careful, as this is not reversible and you do not have a second chance to change your mind.

You may exit the program using the quit command. When doing so, all variables are lost. However, invoking the command save filename before exiting, causes all variables to be written to a binary file called filename.mat. When we start MATLAB again, we may retrieve the information in this file with the command load filename. We can also create an ascii (text) file containing the entire MATLAB session, if we use the command diary filename at the beginning and at the end of our session. This will create a text file called filename (with no extension) that can be edited with any text editor, printed out etc. This file will include everything we typed into MATLAB during the session (including error messages but excluding plots). We could also use the command save filename at the end
of our session to create the binary file described above as well as the text file that includes our work.

One last command to mention before we start learning some more interesting things about MATLAB, is the `help` command. This provides help for any existing MATLAB command or any quantity known to MATLAB. (Some would argue this is the most useful command.) Let us try it on the command `ans`.

```matlab
>> help ans
ans Most recent answer.
    ans is the variable created automatically when expressions
    are not assigned to anything else. ANSWer.

    Reference page for ans
```

Usually, the examples included in MATLAB’s help, are very illustrative. Try using the command `help` on itself!

On a PC, `help` is also available from the Window Menus. Sometimes it is easier to look up a command from the list provided there, instead of using the command line `help`.

### 2.2 Vectors and matrices

We have already seen how to define a vector and assign a variable name to it. Often it is useful to define vectors (and matrices) that contain equally spaced entries. This can be done by specifying the first entry, an increment, and the last entry. MATLAB will automatically figure out how many entries you need and include their values. For example, to create a vector whose entries are 0, 1, 2, 3, …, 7, 8, you can type

```matlab
>> u = [0:8]
```

```
0     1     2     3     4     5     6     7     8
```

Here we specified the first entry 0 and the last entry 8, separated by a colon ( : ). MATLAB automatically filled-in the (omitted) entries using the (default) increment 1. You could also specify an increment, as is done next to obtain a vector `v` whose entries are 0, 2, 4, 6, and 8:

```matlab
>> v = [0:2:8]
```

```
0     2     4     6     8
```

Here we specified the first entry 0, the increment value 2, and the last entry 8. The two colons ( : ) “tell” MATLAB to fill in the (omitted) entries using the specified increment value. If, instead of the step-size, we are interested in the specific number of points being created between the (initial and final) values, it is easier to use the command `linspace`. It takes as
input the initial and final values, as well as the number of desired points. If the last input is omitted, the default value of 101 points is used. So the command below produces a subdivision of [0, 1] with 6 points.

```matlab
>> x = linspace(0,1,6)
x =
     0    0.2000    0.4000    0.6000    0.8000    1.0000
```

MATLAB also allows you to look at specific parts of the vector. If you want, for example, to only look at the first 3 entries in the vector \( v \), you can use the same notation you used to create the vector:

```matlab
>> v(1:3)
an =
     0     2     4
```

Note that we used parentheses, instead of brackets, to refer to the entries of the vector. Since we omitted the increment value, MATLAB automatically assumes that the increment is 1. The following command lists the first 4 entries of the vector \( v \), using the increment value 2.

```matlab
>> v(1:2:4)
an =
     0     4
```

Defining a matrix, is similar to defining a vector. To define a matrix \( A \), you can treat it like a column of row vectors. That is, you enter each row of the matrix as a row vector (remember to separate the entries either by commas or spaces) and you separate the rows by semicolons ( ; ). For example,

```matlab
>> A = [1 2 3; 3 4 5; 6 7 8]
A =
     1     2     3
     3     4     5
     6     7     8
```

We can avoid separating each row with a semicolon if we use a carriage return instead. In other words, we could have defined \( A \) as follows

```matlab
>> A = [1 2 3; 3 4 5; 6 7 8]
A =
     1     2     3
     3     4     5
     6     7     8
```

which is perhaps closer to the way we would have defined \( A \) by hand using the linear algebra
notation. You can refer to a particular entry in a matrix, by using parentheses. For example, the number 5 lies in the 2\textsuperscript{nd} row, 3\textsuperscript{rd} column of \( A \), thus

\begin{verbatim}
>> A(2,3)
an =
      5
\end{verbatim}

The order of rows and columns follows the convention adopted in the linear algebra notation. This means that \( A(2, 3) \) refers to the number 5 in the above example and \( A(3, 2) \) refers to the number 7, which is in the 3\textsuperscript{rd} row, 2\textsuperscript{nd} column. Note MATLAB’s response when we ask for the entry in the 4\textsuperscript{th} row, 1\textsuperscript{st} column:

\begin{verbatim}
>> A(4,1)
Index exceeds matrix dimensions.
\end{verbatim}

As expected, we get an error message. Since \( A \) is a 3-by-3 matrix, there is no 4\textsuperscript{th} row and MATLAB realizes that. The error messages that we get from MATLAB can be quite informative when trying to find out what went wrong. In this case MATLAB told us exactly what the problem was.

We can “extract” submatrices using a similar notation as above. For example, to obtain the submatrix that consists of the first two rows and last two columns of \( A \) we type

\begin{verbatim}
>> A(1:2,2:3)
an =
   2  3
   4  5
\end{verbatim}

We could even extract an entire row or column of a matrix, using the colon (:) as follows. Suppose we want to get the 2\textsuperscript{nd} column of \( A \). We basically want the elements \[ [A(1,2) \ A(2,2) \ A(3,2)] \]. We type

\begin{verbatim}
>> A(:,2)
an =
   2
   4
   7
\end{verbatim}

where the colon was used to tell MATLAB that all the rows are to be used. The same can be done when we want to extract an entire row, say the 3\textsuperscript{rd} one.

\begin{verbatim}
>> A(3,:)
an =
   6  7  8
\end{verbatim}

Let us define another matrix \( B \) and two vectors \( s \) and \( t \) that will be used in what follows.
>> B = [
-1 3 10
-9 5 25
0 14 2]
B =
   -1    3   10
   -9    5   25
    0   14    2

>> s = [-1 8 5]
s =
   -1    8    5

>> t = [7;0;11]
t =
   7
   0
  11

The real power of MATLAB is the ease in which you can manipulate vectors and matrices. For example, to subtract 1 from every entry in the matrix A we type

>> A-1
ans =
   0    1    2
   2    3    4
   5    6    7

It is just as easy to add (or subtract) two compatible matrices (i.e. matrices of the same size).

>> A+B
ans =
   0    5   13
  -6    9   30
   6   21   10

The same is true for vectors, as is illustrated in the examples that follow.

>> s-t'
ans =
  -8    8   -6

>> s-t
??? Error using ==> -
Matrix dimensions must agree.

>> B*s
Error using  *
Inner matrix dimensions must agree.

>> B*t
ans =
  103
  212
  22

Another important operation that MATLAB can perform with ease, is solving linear systems in matrix form. We recall that an $n$-by-$n$ matrix $M$ is invertible if the only solution to the system $Mx = 0$, is $x = 0$. Now if $M$ is an invertible matrix and $b$ is a $n$-by-$1$ vector then

$$x = M\backslash b$$

is the solution of $Mx = b$. Let us illustrate this with $M = B$ and $b = t$.

>> x = B\t
x =
  2.4307
  0.6801
  0.7390

$x$ is the solution of $Bx = t$ as can be seen in the multiplication below.

>> B*x
ans =
  7.0000
 -0.0000
 11.0000

Since $x$ does not consist of integers, it is appropriate to mention here the command \texttt{format long}. MATLAB displays only four digits beyond the decimal point of a real number unless we use the command \texttt{format long}, which tells MATLAB to display more digits.

>> format long
>> x
x =
  2.430715935334873
  0.680138568129330
  0.739030023094688

On a PC the command \texttt{format long} can also be used through the Window Menus. (There are other types of formats, e.g. \texttt{format bank}, so make sure you get help on the command \texttt{format}.)
There are many times when we want to perform an operation to every entry in a vector or matrix. MATLAB will allow us to do this with “element-wise” operations. For example, suppose you want to multiply each entry in the vector \( s \) with itself. In other words, suppose you want to obtain the vector \( s^2 = [s(1) \times s(1), \ s(2) \times s(2), \ s(3) \times s(3)] \). The command \( s \times s \) will not work due to incompatibility (as already seen above). What is needed here is to tell MATLAB to perform the multiplication element-wise. This is done with the symbols ".*". In fact, you can put a period in front of most operators to tell MATLAB that you want the operation to take place element-wise.

```matlab
>> s.*s
ans =
1  64  25
```

The symbol ".^" can also be used since we are after all raising \( s \) to a power. (The period is needed here as well.)

```matlab
>> s.^2
ans =
1  64  25
```

The table below summarizes the operators that are available in MATLAB.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
</tr>
<tr>
<td>^</td>
<td>Power</td>
</tr>
<tr>
<td>'</td>
<td>Transpose</td>
</tr>
<tr>
<td>\</td>
<td>Left division</td>
</tr>
<tr>
<td>/</td>
<td>Right division</td>
</tr>
</tbody>
</table>

Remember that the multiplication, power and division operators can be used in conjunction with a period to specify an element-wise operation.

### Exercises

Create a diary session called `sec2_2` in which you should complete the following exercises. Define

\[
A = \begin{bmatrix} 2 & 9 & 0 & 0 \\ 0 & 4 & 1 & 4 \\ 7 & 5 & 5 & 1 \\ 7 & 8 & 7 & 4 \end{bmatrix}, \quad b = \begin{bmatrix} -1 \\ 6 \\ 0 \\ 9 \end{bmatrix}, \quad a = \begin{bmatrix} 3 & -2 & 4 & -5 \end{bmatrix}
\]
1. Calculate the following (when defined)
   (a) $A \cdot b$  (b) $a + 4$  (c) $b \cdot a$  (d) $a \cdot b^T$  (e) $A \cdot a^T$

2. Explain any differences between the answers that MATLAB gives when you type in $A*A$, $A^2$ and $A.^2$.

3. What is the command that isolates the submatrix consisting of the 2nd to 3rd rows of the matrix $A$?

4. Solve the linear system $A \cdot x = b$ for $x$. Check your answer by multiplication.

Edit your text file to delete any errors (or typos) and obtain a readable printout.

### 2.3 Built-in functions

There are numerous built-in functions (i.e. commands) in MATLAB. We will mention a few of them in this section by separating them into categories.

#### Scalar Functions

Certain MATLAB functions are essentially used on scalars, but operate element-wise when applied to a vector. Some of them are summarized in the table below.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sin</td>
<td>trigonometric sine</td>
</tr>
<tr>
<td>cos</td>
<td>trigonometric cosine</td>
</tr>
<tr>
<td>tan</td>
<td>trigonometric tangent</td>
</tr>
<tr>
<td>asin</td>
<td>trigonometric inverse sine (arcsine)</td>
</tr>
<tr>
<td>acos</td>
<td>trigonometric inverse cosine (arccosine)</td>
</tr>
<tr>
<td>atan</td>
<td>trigonometric inverse tangent (arctangent)</td>
</tr>
<tr>
<td>exp</td>
<td>exponential</td>
</tr>
<tr>
<td>log</td>
<td>natural logarithm</td>
</tr>
<tr>
<td>abs</td>
<td>absolute value</td>
</tr>
<tr>
<td>sqrt</td>
<td>square root</td>
</tr>
<tr>
<td>rem</td>
<td>remainder of division</td>
</tr>
<tr>
<td>round</td>
<td>round towards nearest integer</td>
</tr>
<tr>
<td>floor</td>
<td>round towards negative infinity</td>
</tr>
<tr>
<td>ceil</td>
<td>round towards positive infinity</td>
</tr>
</tbody>
</table>

Even though we will illustrate some of the above commands in what follows, it is strongly recommended to get help on all of them to find out exactly how they are used.

The trigonometric functions take as input radians. Since MATLAB uses $\pi$ for the number $\pi = 3.1415\ldots$, we have
\[
\text{>> sin(pi/2)} \\
\text{ans =} \\
\text{1} \\
\text{>> cos(pi/2)} \\
\text{ans =} \\
\text{6.1232e-17}
\]

The sine of \(\pi/2\) is indeed 1 but we expected the cosine of \(\pi/2\) to be 0. Well, remember that MATLAB is a numerical package and the answer we got (in scientific notation, see format) is very close to 0 (6.1232e-017 = 6.1232\times10^{-17} \approx 0). Since the \texttt{exp} and \texttt{log} commands are straightforward to use, let us illustrate some of the other commands. The \texttt{rem} command gives the remainder after division. So, the remainder of 12 divided by 4 is zero

\[
\text{>> rem(12,4)}
\]
\[
\text{ans =}
\]
\[
0
\]

and the remainder of 12 divided by 5 is 2.

\[
\text{>> rem(12,5)}
\]
\[
\text{ans =}
\]
\[
2
\]

The \texttt{floor}, \texttt{ceil} and \texttt{round} commands are illustrated below.

\[
\text{>> floor(1.4)}
\]
\[
\text{ans =}
\]
\[
1
\]

\[
\text{>> ceil(1.4)}
\]
\[
\text{ans =}
\]
\[
2
\]

\[
\text{>> round(1.4)}
\]
\[
\text{ans =}
\]
\[
1
\]

Keep in mind that all of the above commands can be used on vectors with the operation taking place element-wise. For example, if \(x = [0, 0.1, 0.2, \ldots, 0.9, 1]\), then \(y = \exp(x)\) will produce another vector \(y\), of the same length, whose entries are given by \(y = [e^0, e^{0.1}, \ldots, e^1]\).

\[
\text{>> x = [0:0.1:1]} \\
x =
\]
\[
\begin{array}{cccccc}
0 & 0.1000 & 0.2000 & 0.3000 & 0.4000 & 0.5000 \\
0.6000 & 0.7000 & 0.8000 & 0.9000 \\
\end{array}
\]

\[
\text{Column 11} \\
1.0000 \\
\]

\[
\text{>> y = exp(x)}
\]
This is extremely useful when plotting data. See Section 2.4 ahead for more details on plotting. Also, note that MATLAB displayed the results as 1-by-11 matrices (i.e. row vectors of length 11). Since there was not enough space on one line for the vectors to be displayed, MATLAB reports the column numbers.

**Vector Functions**

Other MATLAB functions operate essentially on vectors returning a scalar value. Some of these functions are given in the table below.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>max</td>
<td>largest component</td>
</tr>
<tr>
<td>min</td>
<td>smallest component</td>
</tr>
<tr>
<td>length</td>
<td>length of a vector</td>
</tr>
<tr>
<td>sort</td>
<td>sort in ascending order</td>
</tr>
<tr>
<td>sum</td>
<td>sum of elements</td>
</tr>
<tr>
<td>prod</td>
<td>product of elements</td>
</tr>
<tr>
<td>median</td>
<td>median value</td>
</tr>
<tr>
<td>mean</td>
<td>mean value</td>
</tr>
<tr>
<td>std</td>
<td>standard deviation</td>
</tr>
</tbody>
</table>

Once again, it is strongly suggested to get help on all the above commands. Some are illustrated next. Let \( z \) be the following row vector.

\[ \text{\texttt{>> z = [0.9347, 0.3835, 0.5194, 0.8310]}} \]

\[ z = \\
0.9347 \quad 0.3835 \quad 0.5194 \quad 0.8310 \]

Then

\[ \text{\texttt{>> max(z)}} \]
\[ \text{ans =} \]
\[ 0.9347 \]

\[ \text{\texttt{>> min(z)}} \]
\[ \text{ans =} \]
\[ 0.3835 \]
The above (vector) commands can also be applied to a matrix. In this case, they act in a column-by-column fashion to produce a row vector containing the results of their application to each column. The example below illustrates the use of the above (vector) commands on matrices. Suppose we wanted to find the maximum element in the following matrix.

\[
M = \begin{bmatrix}
0.7012 & 0.2625 & 0.3282 \\
0.9103 & 0.0475 & 0.6326 \\
0.7622 & 0.7361 & 0.7564
\end{bmatrix}
\]

If we use the `max` command on M, we get the maximum of each column, stored in a row vector (we note that vector functions act on matrices in a column-by-column fashion).

\[
\text{max}(M) \quad \Rightarrow \quad \text{ans} = 0.9103 \quad 0.7361 \quad 0.7564
\]

To isolate the largest element of M, we use the `max` command on the above row vector. Taking advantage of the fact that MATLAB assigns the variable name `ans` to the answer we obtained, we can simply type

\[
\text{max}(\text{ans}) \quad \Rightarrow \quad \text{ans} = 0.9103
\]

The two steps above can be combined into one, as follows:

\[
\text{max}(\text{max}(M)) \quad \Rightarrow \quad \text{ans} = 0.9103
\]

Combining MATLAB commands can be very useful when programming complex algorithms where we do not wish to see or access intermediate results. More on this, and other programming features of MATLAB in Section 3 ahead.
Matrix Building Functions

Much of MATLAB’s power comes from its matrix functions. These can be further separated into two sub-categories. The first one consists of convenient matrix building functions, some of which are given in the table below.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>eye</code></td>
<td>identity matrix</td>
</tr>
<tr>
<td><code>zeros</code></td>
<td>matrix of zeros</td>
</tr>
<tr>
<td><code>ones</code></td>
<td>matrix of ones</td>
</tr>
<tr>
<td><code>diag</code></td>
<td>extract diagonal of a matrix or create diagonal matrices</td>
</tr>
<tr>
<td><code>triu</code></td>
<td>upper triangular part of a matrix</td>
</tr>
<tr>
<td><code>tril</code></td>
<td>lower triangular part of a matrix</td>
</tr>
<tr>
<td><code>rand</code></td>
<td>randomly generated matrix</td>
</tr>
</tbody>
</table>

Make sure you ask for help on all the above commands.

To create the identity matrix of size 4 (i.e. a square 4-by-4 matrix with ones on the main diagonal and zeros everywhere else) we use the command `eye`.

```matlab
>> eye(4,4)
ans =
    1     0     0     0
    0     1     0     0
    0     0     1     0
    0     0     0     1
```

The numbers in parenthesis indicates the size of the matrix. When creating square matrices, we can specify only one input referring to size of the matrix. For example, we could have obtained the above identity matrix by simply typing `eye(4)`. The same is true for the matrix building functions below.

In a similar way, the command `zeros` creates a matrix of zeros and the command `ones` creates a matrix of ones.

```matlab
>> zeros(2,3)
ans =
    0     0     0
    0     0     0
```

```matlab
>> ones(2)
ans =
    1     1
    1     1
```

We can create a randomly generated matrix using the `rand` command. (The entries will be uniformly distributed between 0 and 1.)

```matlab
>> C = rand(5,4)
```
The commands \texttt{triu} and \texttt{tril}, extract the upper and lower part of a matrix, respectively. Let us try them on the matrix \( C \) defined above.

\begin{verbatim}
>> triu(C)
ans =
0.8147 0.0975 0.1576 0.1419
0 0.2785 0.9706 0.4218
0 0 0.9572 0.9157
0 0 0 0.7922
\end{verbatim}

\begin{verbatim}
>> tril(C)
ans =
0.8147 0 0 0
0.9058 0.2785 0 0
0.1270 0.5469 0.9572 0
0.9134 0.9575 0.4854 0.7922
0.6324 0.9649 0.8003 0.9595
\end{verbatim}

Once the extraction took place, the “empty” positions in the new matrices are automatically filled with zeros. As mentioned earlier, the command \texttt{diag} has two uses. The first use is to extract a diagonal of a matrix, e.g. the main diagonal. Suppose \( D \) is the matrix given below. Then, \texttt{diag(D)} produces a column vector, whose components are the elements of \( D \) that lie on its main diagonal.

\begin{verbatim}
>> D = [0.9092 0.5045 0.9866
 0.0606 0.5163 0.4940
 0.9047,0.3190,0.2661];

>> diag(D)
ans =
0.9092
0.5163
0.2661
\end{verbatim}

The second use is to create diagonal matrices. For example,

\begin{verbatim}
>> diag([0.9092;0.5163;0.2661])
\end{verbatim}
creates a diagonal matrix whose non-zero entries are specified by the vector given as input. (A short cut to the above construction is \texttt{diag(diag(D))}). This command is not restricted to the main diagonal of a matrix; it works on ‘off diagonals’ as well. See \texttt{help diag} for more information.

Let us now summarize some of the commands in the second sub-category of matrix functions.

<table>
<thead>
<tr>
<th>size</th>
<th>size of a matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>det</td>
<td>determinant of a square matrix</td>
</tr>
<tr>
<td>inv</td>
<td>inverse of a matrix</td>
</tr>
<tr>
<td>rank</td>
<td>rank of a matrix</td>
</tr>
<tr>
<td>rref</td>
<td>reduced row echelon form</td>
</tr>
<tr>
<td>eig</td>
<td>eigenvalues and eigenvectors</td>
</tr>
<tr>
<td>poly</td>
<td>characteristic polynomial</td>
</tr>
<tr>
<td>norm</td>
<td>norm of matrix (1-norm, 2-norm, (\infty)-norm)</td>
</tr>
<tr>
<td>cond</td>
<td>condition number in the 2-norm</td>
</tr>
<tr>
<td>lu</td>
<td>LU factorization</td>
</tr>
<tr>
<td>qr</td>
<td>QR factorization</td>
</tr>
<tr>
<td>chol</td>
<td>Cholesky decomposition</td>
</tr>
<tr>
<td>svd</td>
<td>singular value decomposition</td>
</tr>
</tbody>
</table>

Don’t forget to get \texttt{help} on the above commands. To illustrate a few of them, define the following matrix.

\[
\begin{bmatrix}
9 & 7 & 0 \\
0 & 8 & 6 \\
7 & 1 & -6
\end{bmatrix}
\]

\[
\text{>> } A = \begin{bmatrix} 9,7,0;0,8,6;7,1,-6 \end{bmatrix}
\]

\[
A =
\begin{bmatrix}
9 & 7 & 0 \\
0 & 8 & 6 \\
7 & 1 & -6
\end{bmatrix}
\]

\[
\text{>> size}(A)
\]

\[
\text{ans} = 
\begin{bmatrix}
3 & 3
\end{bmatrix}
\]

\[
\text{>> det}(A)
\]

\[
\text{ans} = 
\begin{bmatrix}
-192.0000
\end{bmatrix}
\]

Since the determinant is not zero, the matrix is invertible.
>> inv(A)

ans =
   0.2812   -0.2187   -0.2187
   -0.2187    0.2812    0.2812
   0.2917   -0.2083   -0.3750

We can check our result by verifying that $AA^{-1} = A^{-1}A = I$.

>> A*inv(A)

ans =
   1.0000         0         0
   0.0000    1.0000
   0.0000         0    1.0000

>> inv(A)*A

ans =
   1.0000    0.0000    0.0000
   0.0000    1.0000
   -0.0000    0.0000    1.0000

Let us comment on why MATLAB uses both 0’s and ±0.0000’s in the answer above. Recall that we are dealing with a numerical package that uses numerical algorithms to perform the operations we ask for. Hence, the use of floating point (vs. exact) arithmetic causes the “discrepancy” in the results.

The eigenvalues and eigenvectors of A (i.e. the numbers $\lambda$ and nonzero vectors $x$ that satisfy $Ax = \lambda x$) can be obtained through the eig command.

>> eig(A)

ans =
   -4.8055
   12.6462
   3.1594

produces a column vector with the eigenvalues and

>> [X,D]=eig(A)

X =
   0.2103    0.8351    0.6821
   -0.4148    0.4350   -0.5691
   0.8853    0.3368    0.4592

D =
   -4.8055         0         0
         0  12.6462         0
         0         0    3.1594
produces a diagonal matrix $D$ with the eigenvalues on the main diagonal, and a full matrix $X$ whose columns are the corresponding (normalized) eigenvectors.

**Exercises**

Create a diary session called sec2_3 in which you should complete the following exercises using MATLAB commands. When applicable, use the matrix $A$ and the vectors $b, a$ that were defined in the previous section’s exercises.

1. Construct a randomly generated 2-by-2 matrix of positive integers less than 99.

2. Find the maximum and minimum elements in the matrix $A$.

3. Sort the values of the vector $b$ in ascending order.

4. (a) Find the eigenvalues and eigenvectors of the matrix $B = A^{-1}$. Store the eigenvalues in a column vector you should name $\lambda$.
   (b) With $I$ the 4-by-4 identity matrix, calculate the determinant of the matrix $B - \lambda(j)I$, for $j = 1, 2, 3, 4$. What do you observe?

**2.4 Plotting**

We end our discussion on the basic features of MATLAB by introducing the commands for data visualization (i.e. plotting). By typing `help plot` you can see the various capabilities of this important command for two-dimensional plotting, some of which will be illustrated below.

If $x$ and $y$ are two vectors of the same length then `plot(x, y)` plots $x$ versus $y$.

For example, to obtain the graph of $y = \cos(x)$ from $-\pi$ to $\pi$, we can first define the vector $x$ with components equally spaced numbers between $-\pi$ and $\pi$:

```matlab
>> x = linspace(-pi,pi);  
We placed a semicolon at the end of the input line to avoid seeing the (long) output.
Next, we define the vector $y$
```  
(using a semicolon again) and we ask for the plot
```matlab
>> plot(x,y)
```
At this point a new window will open on our desktop in which the graph (as seen below) will
It is good practice to label the axis on a graph and if applicable indicate what each axis represents. This can be done with the `xlabel` and `ylabel` commands.

```matlab
>> xlabel('x')
>> ylabel('y=cos(x)')
```

Inside parentheses, and enclosed within single quotes, we type the text that we wish to be displayed along the $x$ and $y$ axis, respectively. We could even put a title on top using

```matlab
>> title('Graph of cosine from $-$ \pi to \pi')
```

as long as we remember to enclose the text in parentheses within single quotes. The back-slash (\) in front of pi allows the user to take advantage of certain LaTeX commands. If you are not familiar with the mathematical typesetting software LaTeX (and its commands), ignore the previous command and simply type
Both graphs are shown below. These commands can be invoked even after the plot window has been opened and MATLAB will make all the necessary adjustments to the display.

```
>> title('Graph of cosine from -pi to pi')
```

Various line types, plot symbols and colors can be used. If these are not specified (as in the case above) MATLAB will assign (and cycle through) the default ones, as given below.

```
<table>
<thead>
<tr>
<th>Color</th>
<th>Line Type</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>blue</td>
<td>.</td>
</tr>
<tr>
<td>g</td>
<td>green</td>
<td>o</td>
</tr>
<tr>
<td>r</td>
<td>red</td>
<td>x</td>
</tr>
<tr>
<td>c</td>
<td>cyan</td>
<td>+</td>
</tr>
<tr>
<td>m</td>
<td>magenta</td>
<td>*</td>
</tr>
<tr>
<td>y</td>
<td>yellow</td>
<td>s</td>
</tr>
<tr>
<td>k</td>
<td>black</td>
<td>d</td>
</tr>
<tr>
<td>w</td>
<td>white</td>
<td>v</td>
</tr>
</tbody>
</table>

(b) point, (g) circle, (r) x-mark, (c) plus, (m) star, (y) square, (k) diamond, (w) triangle (down), (v) triangle (up), (l) triangle (left), (g) triangle (right), (p) pentagram, (h) hexagram)
```

So, to obtain the same graph but in green, we type

```
>> plot(x,y,'g')
```

where the third argument indicating the color, appears within single quotes. We could get a dashed line instead of a solid one by typing

```
>> plot(x,y,'--')
```

or even a combination of line type and color, say a blue dotted line by typing

```
>> plot(x,y,'b:')
```

Multiple curves can appear on the same graph. If for example we define another vector

```
>> z = sin(x);
```

we can get both graphs on the same axis, distinguished by their line type, using
>> plot(x, y, 'r--', x, z, 'b: ')

The resulting graph can be seen below, with the red dashed line representing $y = \cos(x)$ and the blue dotted line representing $z = \sin(x)$.

![Graph with red dashed line and blue dotted line](image)

When multiple curves appear on the same axis, it is a good idea to create a *legend* to label and distinguish them. The command `legend` does exactly this.

>> legend('cos(x)', 'sin(x)')

The text that appears within single quotes as input to this command, represents the legend labels. We must be consistent with the ordering of the two curves, so since in the `plot` command we asked for cosine to be plotted before sine, we must do the same here.

![Legend with 'cos(x)' and 'sin(x)'](image)

At any point during a MATLAB session, you can obtain a hard copy of the current plot by either issuing the command `print` at the MATLAB prompt, or by using the command menus on the plot window. In addition, MATLAB plots can be copied and pasted (as pictures) in your favorite word processor (e.g. Microsoft Word). This can be achieved using the Edit menu on the figure window. Another nice feature that can be used in conjunction with `plot` is the command `grid`, which places grid lines to the current axis (just like you
have on graphing paper). Type `help grid` for more information. Other commands for data visualization that exist in MATLAB include

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>subplot</td>
<td>create an array of (tiled) plots in the same window</td>
</tr>
<tr>
<td>loglog</td>
<td>plot using log-log scales</td>
</tr>
<tr>
<td>semilogx</td>
<td>plot using log scale on the x-axis</td>
</tr>
<tr>
<td>semilogy</td>
<td>plot using log scale on the y-axis</td>
</tr>
<tr>
<td>surf</td>
<td>3-D shaded surface graph</td>
</tr>
<tr>
<td>contour</td>
<td>Contour plot</td>
</tr>
<tr>
<td>mesh</td>
<td>3-D mesh surface</td>
</tr>
</tbody>
</table>

It is left to the reader to further investigate the above commands through MATLAB’s `help`. We illustrate here how to obtain one of the surface pictures on the cover of this guide:

```matlab
>> [x,y] = meshgrid(-3:.1:3,-3:.1:3);
>> z = 3*(1-x).^2.*exp(-(x.^2) - (y+1).^2) ... 
  - 10*(x/5 - x.^3 - y.^5).*exp(-x.^2-y.^2) ... 
  - 1/3*exp(-(x+1).^2 - y.^2);
>> surf(z)
>> xlabel('x')
>> ylabel('y')
>> zlabel('z')
>> title('Peaks')
```

Type `help meshgrid`, `help surf` and `help peaks` for more information on the above surface.

**Exercise**

Obtain a hard copy of the plot of the functions $f(x) = x^2$, $g(x) = x^3$ for $x = -1, \ldots, 1$ on the same axis. Label the $x$ and $y$ axes and create a legend indicating which graph is which.
3. PROGRAMMING IN MATLAB

3.1 M-files: Scripts and functions

To take advantage of MATLAB’s full capabilities, we need to know how to construct long (and sometimes complex) sequences of statements. This can be done by writing the commands in a file and calling it from within MATLAB. Such files are called “m-files” because they must have the filename extension “.m”. This extension is required in order for these files to be interpreted by MATLAB. There are two types of m-files: script files and function files. Script files contain a sequence of usual MATLAB commands, that are executed (in order) once the script is called within MATLAB. For example, if such a file has the name compute.m, then typing the command compute at the MATLAB prompt will cause the statements in that file to be executed. Script files can be very useful when entering data into a matrix. Function files, on the other hand, play the role of user defined commands that often have input and output. You can create your own commands for specific problems this way, which will have the same status as other MATLAB commands. Let us give a simple example. The text below is saved in a file called log3.m and it is used to calculate the base 3 logarithm of a positive number. The text file can be created in a variety of ways, for example using the built-in MATLAB editor through the command edit (available with MATLAB 5.0 and above), or your favorite (external) text editor (e.g. Notepad or Wordpad in Microsoft Windows). You must make sure that the filename has the extension “.m”

```
function [a] = log3(x)
% [a] = log3(x) - Calculates the base 3 logarithm of x.
a = log(abs(x))/log(3);
% End of function
```

Using this function within MATLAB to compute \( \log_{3}(5) \), we get

```
>> log3(5)
ans =
    1.4650
```

Let us explain a few things related to the syntax of a function file. Every MATLAB function begins with a header, which consists of the following:

(a) the word function,
(b) the output(s) in brackets, (the variable \( a \) in the above example)
(c) the equal sign,
(d) the name of the function, which must match the function filename (\texttt{log3} in the above example) and
(e) the input(s) (the variable \( x \) in the above example).
Any statement that appears after a “%” sign on a line is ignored by MATLAB and plays the role of comments in the subroutine. Comments are essential when writing long functions or programs, for clarity. In addition, the first set of comments after the header in a function serves as on-line help. For example, see what happens when we type

```
>> help log3
```

```
[a] = log3(x) - Calculates the base 3 logarithm of x.
```

MATLAB gave us “help” on the function we defined, the text that we included after the header in the file. Finally, the algorithm used to calculate the base 3 logarithm of a given number, is based on the formula $\log_3(x) = \frac{\ln(|x|)}{\ln(3)}$. Since the logarithm of a negative number is undefined, we use the absolute value for “safety”. Also, note that we have allowed for a vector to be passed as input, by using element-wise division in the formula.

During a MATLAB session, we may call a function just like we did in the above example, provided the file is saved in the current (working) directory. This is the reason why in the beginning of this guide we suggested that you should create a working directory and switch to that directory from within MATLAB. It should be noted that both types of m-files can reference other m-files, including themselves in a recursive way.

**Exercise**

Write a `script` m-file called `rand_int.m` that once called within MATLAB gives a random, positive integer less than 100.

### 3.2 Loops

We will now cover some commands for creating loops, which are not only used in writing m-files, but in regular MATLAB sessions as well. The examples that we will give will include both situations. The two types of loops that we will discuss are “for” and “while” loops. Both loop structures in MATLAB start with a keyword such as `for`, or `while` and they end with the word `end`. The “for” loop allows us to repeat certain commands. If you want to repeat some action in a predetermined way, you can use the “for” loop. The “for” loop will loop around some statement, and you must tell MATLAB where to start and where to end. For example,

```
>> for j = 1:4
    j+2
end
```

```
ans =
3
```
looped through the numbers 1, ..., 4 and every time printed the current number plus 2. Enclosed between the `for` and `end`, you can have multiple statements just like in the example below. Here, we define the vector $x = [1, 2, ..., 10]$ and we calculate $x^2 = [1^2, 2^2, ..., 10^2]$, which we name $x_2$. The semicolon at the end of the inner statement in the loop suppresses the printing of unwanted intermediate results.

```matlab
>> x = 1:10
x =
     1     2     3     4     5     6     7     8     9    10
>> for i = 1:10
    x2(i) = x(i)^2;
end
>> x2
x2 =
     1     4     9    16    25    36    49    64    81   100
```

Even though for loops are convenient to use in certain cases, they are not always the most efficient way to perform an operation. In the above example, we would have been better off using

```matlab
>> x2 = x.^2
x2 =
     1     4     9    16    25    36    49    64    81   100
```

instead. There are occasions, however, where the “vectorized” notation of MATLAB cannot help us perform the operations we want, and loops are needed, despite the fact that they are not as efficient. Nested loops can also be created. In the following example, we calculate the square of the entries in a matrix. (This is again not efficient, but it is used for illustration purposes only.)

```matlab
>> A = [1,5,-3;4,0;1,6,9]
A =
     1     5    -3
     2     4     0
    -1     6     9
>> for i=1:3
    for j=1:3
        A2(i,j) = A(i,j)^2;
    end
end
```
Again, we mention that a more efficient way of performing the above is

```matlab
>> A2=A.^2
A2 =
1  25  9
4  16  0
1  36  81
```

For a more realistic (but still naive) example, consider the m-file `gaussel.m`, which performs (naive) Gaussian elimination (and back substitution) to solve the square system $Ax=b$.

```matlab
function [x] = gaussel(A,b)

% [x] = gaussel(A,b)
% This subroutine will perform Gaussian elimination
% and back substitution to solve the system Ax = b.
% b - vector for the right hand side
% OUTPUT : x - the solution vector.
N = length(A);

% Perform Gaussian Elimination
for j=2:N,
    for i=j:N,
        m = A(i,j-1)/A(j-1,j-1); %define the pivot
        A(i,:) = A(i,:)-A(j-1,:)*m; %Row_i <-> Row_i - m*Row_{j-1}
        b(i) = b(i)-m*b(j-1); %Same for vector b
    end
end
% Perform back substitution
x = zeros(N,1);
x(N) = b(N)/A(N,N); %
for j=N-1:-1:1,
    x(j) = (b(j)-A(j,j+1:N)*x(j+1:N))/A(j,j);
end
% End of function
```

To illustrate the use of the above file, we define

```matlab
>> A = [4 3 2 3;1 2 3 6;4 2 2 1;9 9 1 -2]
```
A =
  4  3  2  3
  1  2  3  6
  4  2  2  1
  9  9  1 -2

>> b=[1;0;2;-5]

b =
  1
  0
  2
 -5

(You may check that the matrix A is invertible.)

The solution to \( A \times x = b \) is given by

>> x = gaussel(A,b)

x =
  1.2979
 -1.7660
 -0.0213
  0.3830

Of course, a far more efficient way to solve such a linear system would be through the built-in MATLAB solver. That is, we could have typed \( x = A \backslash b \) to obtain the same answer.

The second type of loop is the “while” loop. The “while” loop repeats a sequence of commands as long as some condition is met. For example, given a number \( n \), the following m-file (exple.m) will display the smallest non-negative integer \( a \) such that \( 2^a \geq n \).

function [a] = exple(n)
% [a] = exple(n)
% a = 0;
while 2^a < n
    a = a + 1;
end
% End of function

>> a = exple(4)
a =
  2

The conditional statement in the “while” loop is what makes it differ from the “for” loop. In the above example we used the conditional statement while \( 2^a < n \) which meant that MATLAB would check to see if this condition is met, and if so proceed with the statement that followed. Such conditional statements are also used in “if” statements that are discussed.
in the next section. To form a conditional statement we use relational operators. The following are available in MATLAB.

<table>
<thead>
<tr>
<th>&lt;</th>
<th>less than</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td>greater than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal</td>
</tr>
<tr>
<td>==</td>
<td>equal</td>
</tr>
<tr>
<td>~=</td>
<td>not equal</td>
</tr>
</tbody>
</table>

Note that “=” is used in assignments and “==” is used in relations. Relations may be connected (or quantified) by the following logical operators.

<table>
<thead>
<tr>
<th>&amp;</th>
<th>and</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>~</td>
<td>not</td>
</tr>
</tbody>
</table>

**Exercise**

The $n$-by-$n$ Hilbert matrix $H$, has as its entries $H_{ij} = 1/(i + j - 1)$, $i,j = 1, 2, \ldots, n$. Create a double “for loop” to generate the 5-by-5 Hilbert matrix and check your answer using the built-in MATLAB command `hilb`.

**3.3 If statement**

There are times when you would like your algorithm/code to make a decision, and the “if” statement is the way to do it. The general syntax in MATLAB is as follows:

```
if relation
    statement(s)
elseif relation % if applicable
    statement(s) % if applicable
else % if applicable
    statement(s) % if applicable
end
```

The logical operators (&, |, ~) could also be used to create more complex relations.

Let us illustrate the “if” statement through a simple example. Suppose we would like to define and plot the piecewise defined function
This is done with the use of the “if” statement in MATLAB as follows. First, we define the “domain” vector $x$ from $-1$ to $1$ using the command `linspace`.

```matlab
>> x = linspace(-1,1);
```

Next, we loop through the values of $x$ and for each one we create the corresponding function value $F$ as a vector.

```matlab
>> for i = 1:length(x)
    if x(i) < 0.5
      F(i) = x(i)^2;
    else
      F(i) = 0.25;
    end
end
```

Finally, we plot the two vectors (using a solid black curve).

```matlab
>> plot(x,F,'k')
```

We note that there is a better way to define the above function, without using for-loops, as seen below. As already mentioned, for-loops can slow down our programs and, if possible, should be avoided. However, this is a beginner’s guide and our goal is to get you started using MATLAB.

```matlab
>> x = linspace(-1,1);
>> index1 = x < 1/2;
>> index2 = x >= 1/2;
>> f(index1) = x(index1).^2;
>> f(index2) = 0.25;
```
As a second example, we would like to write a subroutine that takes as input a square matrix and returns its inverse (if it exists). The m-file below (chk_inv.m) will perform this task for us, and make use of the “if” statement. If the matrix is not square or if it does not have an inverse, the subroutine should print a message letting us know and it will terminate without computing anything. We will also make use of comments within the m-file to make it more readable.

```
function [Ainv] = chk_inv(A)
    % [Ainv] = chk_inv(A)
    % Calculate the inverse of a matrix A, if it exists.
    [m,n] = size(A); % compute the size of the matrix A
    if m~=n % check if A is square
        disp('Matrix is not square.');
        return % quit the function
    elseif det(A)==0 % check if A is singular
        disp('Matrix is singular.');
        return % quit the function
    else
        disp('Inverse exists!');
        Ainv = inv(A); % compute the inverse
    end
    % End of function
```

Here is a sample run of the above program with a random 3-by-3 matrix.

```
>> A = rand(3,3)
A =
   0.8147    0.9134    0.2785
   0.9058    0.6324    0.5469
   0.1270    0.0975    0.9575
>> chk_inv(A)
Inverse exists!
ans =
   -1.9958    3.0630   -1.1690
    2.8839   -2.6919    0.6987
  -0.0291   -0.1320    1.1282
```

It is left to the reader to see what answers other input matrices will produce.

In the above m-file, we used two “new” commands: disp and return. As you can imagine, return simply causes the current program to exit. The command disp takes as
input text enclosed within single quotes and displays it on the screen. See help disp for
more information of this and other text displaying commands.

As a final example, let us write an m-file called fact.m that gives the factorial of a positive
number \( n = 1 \cdot 2 \cdot 3 \cdot 4 \ldots (n - 1) \cdot n \), using the recursive formula \( n! = n \cdot (n - 1) \). This example will
not only illustrate the use of the “if” statement, but that of a recursive function as well.

```matlab
function [N] = fact(n)
% [N] = fact(n)
% Calculate n factorial
if (n == 1) | (n == 0)
    N = 1;
else
    N = fact(n-1);
end
% End of function
```

The above m-file runs satisfactorily for \( n \) small (say less than 50). For larger values, there are
other ways to compute the factorial, such as Stirling’s formula used in the command factorial.

We close this chapter, by introducing anonymous functions which became available in later
versions of MATLAB. Perhaps the most obvious example is defining a mathematical
function (of any number of variables). For instance, \( f(x) = x^2 \cos(x) + e^{-x} \). We write

```matlab
>> f=@(x) (x.^2).*cos(x)+exp(-x);
```

Then, we could get function values by simply typing

```matlab
>> f(1),f(-2),f([-1:1])
```

ans =
   0.9082

ans =
    5.7245

ans =
  3.2586 1.0000  0.9082

The same can be done for functions of several variables, such as \( g(x,y) = x^2 \cos(y) + e^{-x^2-y^2} \):

```matlab
>> g=@(x,y) x.^2.*cos(y)+exp(-x.^2-y.^2);
```
Once we have an anonymously defined function, we may take advantage of commands like fplot, fsurf, fmesh, etc., for producing plots. (Ask MATLAB for help on these commands.) We simply illustrate one of them:

```matlab
>> fsurf(g,[-2,2,-3,4])
```

### Exercises

1. Modify the m-file `log3.m` from Section 3.1, by removing the absolute value within the logarithms (that was used for “safety”). Your function should now check to see if the input is negative or zero, print out a message saying so, and then terminate. If the input is positive then your function should proceed to calculate the logarithm base 3 of the input.

2. Write a function m-file called `div5.m` that takes as input a real number and checks to see if it is divisible by 5. An appropriate message indicating the result should be the output.
4. ADDITIONAL TOPICS

4.1 Polynomials in MATLAB

Even though MATLAB is a numerical package, it has capabilities for handling polynomials. In MATLAB, a polynomial is represented by a vector containing its coefficients in descending order. For instance, the polynomial \( p(x) = x^2 - 3x + 5 \) is represented by the vector \( p = [1, -3, 5] \) and the polynomial \( q(x) = x^4 + 7x^2 - x \) is represented by \( q = [1, 0, 7, -1, 0] \).

MATLAB can interpret any vector of length \( n + 1 \) as an \( n \)th order polynomial. Thus, if your polynomial is missing any coefficients, you must enter zeros in the appropriate place(s) in the vector, as done above.

You can find the value of a polynomial using the command \texttt{polyval}. For example, to find the value of the polynomial \( q \) above at \( x = -1 \), you type

\[
\texttt{>> polyval(q,-1)}
\]

\[
\texttt{ans = 9}
\]

Finding the roots of a polynomial is as easy as entering the following command.

\[
\texttt{>> roots(q)}
\]

\[
\texttt{ans =}
\]

\[
\begin{array}{c}
0.0000 + 0.0000i \\
-0.0712 + 2.6486i \\
-0.0712 - 2.6486i \\
0.1424 + 0.0000i
\end{array}
\]

Note that MATLAB can handle complex numbers as well, with \texttt{i=sqrt(-1)}. This is reflected in the four roots above, two of which are complex. Suppose you want to multiply two polynomials together. Their product is found by taking the convolution of their coefficients. MATLAB’s command \texttt{conv} will do this for you. For example, if \( s(x) = x + 2 \) and \( t(x) = x^2 + 4x + 8 \) then \( z(x) = s(x) \cdot t(x) = x^3 + 6x^2 + 16x + 16 \). In MATLAB, we type

\[
\texttt{>> s = [1 2];}
\]

\[
\texttt{>> t = [1 4 8];}
\]

\[
\texttt{>> z = conv(s,t)}
\]

\[
\texttt{z =}
\]

\[
\begin{array}{cccc}
1 & 6 & 16 & 16
\end{array}
\]

Dividing two polynomials is just as easy. The \texttt{deconv} function will return the remainder as well as the result. Let’s divide \( z \) by \( t \) and see if we get \( s \).
>> [s,r] = deconv(z,t)
s =
    1     2
r =
    0     0     0     0

As you can see, we get (as expected) the polynomial/vector s from before. If s did not divide z exactly, the remainder vector r, would have been something other than zero.

MATLAB can obtain derivatives of polynomials very easily. The command polyder takes as input the coefficient vector of a polynomial and returns the vector of coefficients for its derivative. For example, with \( p(x) = x^2 - 3x + 5 \), as before

```matlab
>> polyder(p)
an =
    2     -3
```

What do you think (in terms of Calculus) the combination of commands polyval(polyder(p),1) gives? How about roots(polyder(p))?

**Exercises**

1. Write a function m-file called polyadd.m that adds two polynomials (of not necessarily the same degree). The input should be the two vectors of coefficients and the output should be a new vector of coefficients representing their sum. (Recent versions of MATLAB include this as a built-in command.)

2. Find the relative maxima and minima (if any) of the polynomial function \( f(x) = x^3 - x^2 - 3x \). Plot the function and the maxima and minima, using a ‘o’ for each minimum and a ‘*’ for each maximum.

**4.2 Numerical Methods**

In this section we mention some useful commands that are used for approximating various quantities of interest. We have already seen that MATLAB can find the roots of a polynomial. Suppose we are interested in finding the root(s) of a general non-linear function. This can be done in MATLAB through the command fzero, which is used to approximate the root of a function of one variable, given an initial guess. We must first define the function either anonymously or in an m-file, and then invoke the fzero command with the name of
that function and an initial guess as input. Consider finding the root(s) of \( f(x) = e^x - x^2 \). We define \( f \) anonymously via

\[
\text{>> } f=@(x) \text{exp}(x) - x.^2;
\]

A plot of the function can prove to be very useful when choosing a good initial guess, since it gives us an idea as to where the root lies.

\[
\text{>> } x=\text{linspace}(-1,3);
\]
\[
\text{>> } \text{plot}(x,f(x))
\]
\[
\text{>> } \text{grid}
\]

We see from the plot that there is one root between \( x = -1 \) and \( x = 0 \). Hence, we choose as an initial guess \(-0.5\) and we type

\[
\text{>> } f\text{zero}(f,-0.5)
\]

\[
\text{ans } = -0.7035
\]

If the function is defined via an m-file, then the name of the function should appear within single quotes as input to \texttt{fzero}. In addition, don’t forget that this is a four-digit approximation to the root. This is seen from

\[
\text{>> } f(-0.7035)
\]

\[
\text{ans } = -6.1957e-005
\]

which is not (quite) zero. Of course, the number of digits beyond the decimal point that are passed as input play an important role. See what happens when we change the format.

\[
\text{>> format long}
\]
\[
\text{>> fzero}(f,-0.5)
\]
ans =
    -0.70346742249839

>> f(-0.70346742249839) \% f(ans) also works
ans =
    0

which means that we have found all the digits correctly.

When a function has more than one root, the value of the initial guess can be changed, in order to obtain approximations for the additional roots.

Another useful command is \texttt{fminbnd}, which works in a similar way as the \texttt{fzero} command, but finds the \textit{minimum} of a function. The command \texttt{fminbnd} requires as input the name of the function (within single quotes if it is defined via an m-file) and an interval over which the minimization will take place. For example, the MATLAB demo function
\[
g(x) = \frac{1}{(x - 0.3)^2 + 0.01} + \frac{1}{(x - 0.9)^2 + 0.04} - 6
\]
is (already) in a file called \texttt{humps.m} and its graph is seen below.

\[
\begin{align*}
\text{>>} & \quad x = \text{linspace}(0,1); \\
\text{>>} & \quad \text{plot}(x,\text{humps}(x))
\end{align*}
\]

We see that there is a local minimum between \(x = 0.5\) and \(x = 1\). So we type

\[
\begin{align*}
\text{>>} & \quad [\text{xmin},\text{ymin}] = \text{fminbnd}('humps',0.5,1) \\
\text{xmin} & = 0.63701067459059 \\
\text{ymin} & = 11.25275412656430
\end{align*}
\]
When multiple minima are present, the endpoints of the interval given as input can be changed, in order to obtain the rest of the minima. How do you think you can find the maximum of a function instead of the minimum? Try it on the function \textit{humps}.

As a final command in this section, we mention the command \texttt{integral} which approximates the value of a definite integral. Once again, the function we wish to integrate must be either defined anonymously or saved in a file and the name, along with the limits of integration must be passed as input to the command \texttt{integral}.

Consider integrating $f(x) = e^x - x^2$ over the interval $x = 0$ to $1$. Since we have already defined this function, we simply type

\begin{verbatim}
>> integral(f,0,1)
ans =  1.384948495125712
\end{verbatim}

How about the integral of $\sin(x)$ from $0$ to $\pi$? We know the answer in this case, and it should be $2$. MATLAB knows the function \texttt{sin}, so without creating any m-file for it, we type

\begin{verbatim}
>> integral(@sin,0,pi)
ans =  2.000000000000000
\end{verbatim}

Notice the @ symbol in front of the (built-in) function \texttt{sin}. We can also control the accuracy of the approximation if we pass as input to the command the relative and absolute \textit{tolerance} we wish to have, i.e. the acceptable error in the approximation measured relatively and absolutely. For example,

\begin{verbatim}
>> integral(@(x) log(x),0,1,'AbsTol',1e-6,'RelTol',1e-3)
ans =  -1.000000175361872
\end{verbatim}

Type \texttt{help integral} for more information.

\section*{4.3 Numerical Solution of ODEs}

MATLAB has excellent ordinary differential equation (ode) solvers, which can provide an approximation to the problem: find $x(t) = [x_1(t), \ldots, x_n(t)]^T$ such that

\[
\begin{align*}
\frac{dx}{dt} &= F(x,t) \\
\frac{dt}{dt} &= x(0) = x_0
\end{align*}
\]
where $F(x, t) = \left[ f_1(x, t), ..., f_n(x, t) \right]^T$, with $f_i, x_0$ given. The above is a system of odes, with an initial condition, commonly referred to as an initial value problem (IVP). In the scalar case the IVP is given by $x'(t) = f(t, x(t)), x(t_0) = x_0$.

Some of the most common commands for the numerical solution of IVPs are: ode23, ode45, ode15s, etc. (Type doc ode23 for more information on that command.) We will illustrate the command ode45, which works as follows:

$$\begin{align*}
[t\_out, x\_out] &= \text{ode45}(\text{odefun, t\_span, x0})
\end{align*}$$

where
- odefun: corresponds to the right hand side of the ode. In the case of systems, $F$ must be defined in an m-file, and in the scalar case $f$ may be defined anonymously.
- t\_span: the vector $[t_0, T]$ (such that $t \in [t_0, T]$) – this must be given using brackets, e.g. [0, 1].
- x0: the initial condition $x_0(= x(t_0))$
- t\_out: the vector with values $t_0, t_1, t_2...$
- x\_out: the vector with the values $x_0, x_1, x_2...$ which approximate the solution at the points $t_i$.

For example, let us use the command for the IVP $x'(t) = x(2 - x), x(0) = 3$:

```matlab
>> f = @(t, x) x.*(2-x);
>> [t, x]=ode45(f, [0,1],3);
```

We graph the approximate solution:

```matlab
>> plot(t, x, '-rx')
>> xlabel('t')
>> ylabel('x')
>> title('Solution to x''(t)=x(2-x), t in [0, 1], x(0)=3')
```
We also consider an example of a system:

\[
\begin{align*}
  x_1'(t) &= -\frac{8}{3} x_1(t) + x_2(t) x_3(t) \\
  x_2'(t) &= -10 x_2(t) + 10 x_3(t) \\
  x_3'(t) &= -x_1(t) x_2(t) + 28 x_3(t) - x_3(t) \\
  x_1(0) &= 20, x_2(0) = 5, x_3(0) = -5
\end{align*}
\]

where \( t \in [0, 12] \). We write the following m-file for the system’s RHS.

```matlab
function [xprime] = odefun(t,x)

% [xprime] = odefun(t,x) -
% This function corresponds to the RHS of the system of ODEs, in which x=[x(1), x(3), x(3)] represents the (vector) of unknown functions, each one being a function of t.

xprime = [-8*x(1)/3+x(2)*x(3); -10*x(2)+10*x(3); -x(2)*x(1)+28*x(3)-x(3)];

% End of m-file odefun.m
```

We type

```matlab
>> [t,x] = ode45(@odefun,[0,12],[20,5,-5]);
```
If we only want the plot of, say $x_2(t)$, we type

```matlab
>> plot(t,x(:,2))
```
5. CLOSING REMARKS

It is our hope that by reading this guide you formed a general idea of the capabilities of
MATLAB, while obtaining a working knowledge of the program. Certain advantages but also
limitations of MATLAB could also be seen through this tutorial, and it is left to the reader to
decide when and if to use this program.

There are numerous other commands, specialized functions and toolboxes that may be of
interest to you. For example, MATLAB’s Symbolic Toolbox includes a “piece” of MAPLE®
(www.maplesoft.com), so that symbolic manipulations can be performed.

A good source of information related to MATLAB, the creator company THE MATHWORKS
INC and their other products is their webpage at www.mathworks.com. It is strongly
recommended that you visit this webpage to see what other publications exist (see below) that
will allow you to enhance your knowledge of MATLAB. Some are more advanced than
others so do not hesitate to talk to your professor for guidance through the (rather) long list.
Hope you enjoyed reading this guide and … keep computing 😊

MATLAB Support:  http://www.mathworks.com/support/

Some recommended MATLAB books: https://www.mathworks.com/support/books.html
APPENDIX
FLOATING POINT ARITHMETIC AND ERRORS

In this appendix we summarize some results about computer arithmetic and the meaning of error, especially as it pertains to MATLAB.

A.1. Computer arithmetic

We will be concerned with floating-point arithmetic as is used by computers. We place the decimal (the radix as it is called) right before the first non-zero digit, and we multiply with the base of our number system, raised to the appropriate power. For example, in the decimal system the number

\[-(0.0005928)_{10}\]

is written as

\[-.598 \times 10^{-2}\]

Similarly, in the binary system, the number

\[(111.001)_{2}\]

is written as

\[.111001 \times 2^{3}\]

In general, in a number system with base \(\beta\), a number in floating point arithmetic has the form

\[\sigma (\cdot a_1 a_2 a_3 \cdots) \beta^e\]  (1)

where

- \(\sigma\) is the sign of the number, with only possible choices +1 and –1
- “.” is the so-called radix point
- \(a_i, i = 1,2,3,\ldots\) are the digits of the number, which satisfy \(\alpha_1 \neq 0, 0 \leq \alpha_i \leq \beta - 1\)
- \(e\) is the exponent which could be any integer

We cannot represent every real number on a computer, since there are restrictions on the number of digits that can be stored, as well as on the exponent. The number \(t\) of significant digits that is stored by the system determines the accuracy. Moreover, the system can handle only a finite number of memory positions for the exponent, so

\[L \leq e \leq U\]

where \(L\) and \(U\) are the lower and upper bounds, respectively. In general,

\[L \approx -U + 1.\]
To summarize, the computer stores numbers as

\[ x = \sigma (a_1 a_2 \cdots a_t) \beta^e \]  

under the restrictions on the exponent \( e \) that we mentioned. The expression

\( (a_1 a_2 \cdots a_t) \beta \)

is called the **mантissa** of the number. The numbers of the form (2) are determined by \( \beta, t, L \)
and \( U \), which in turn depend on the machine. All such numbers are called **machine numbers**. We will use the symbol \( M \) for all machine numbers:

\[ M = M(\beta, t, L, U) \]

**Remarks**

1. Obviously \( M \) is a finite subset of the rational numbers.
2. To get the maximum element of \( M \) in absolute value, we set \( e = U \) and

\[ \alpha_1 = \alpha_2 = \cdots = \alpha_t = (\beta - 1) \]

So, in the binary system (\( \beta = 2 \)) the maximum machine number is

\( (.11 \cdots 1)_2 2^U \)

while for \( \beta = 8 \), the maximum number is

\( (.77 \cdots 7)_8 8^U \)

3. For any base \( \beta \), the minimum machine number in absolute value is

\( (.10 \cdots 0)_\beta \beta^L \)

4. The distance between two successive elements of \( M \) is not constant, but rather it depends on the value of the exponent \( e \). In general, if

\[ x_1 = (a_1 a_2 \cdots a_t) \beta^e \]

is any positive machine number, then the next number is

\[ x_2 = (a_1 a_2 \cdots a_t + \beta^{-t}) \beta^e = (a_1 a_2 \cdots a_t) \beta^e + \beta^{e-t} = x_1 + \beta^{e-t} \Rightarrow x_2 - x_1 = \beta^{e-t} \]  

We note that the distance increases as the exponent \( e \) changes (the precision \( t \) is a machine constant). If we use a decimal machine (\( \beta = 10 \)) with \( t = 5 \), the number right after \( x_1 = (.23012) 10^2 \) is \( x_2 = (.23013) 10^2 \), thus

\[ x_2 - x_1 = (.00001) 10^2 = 10^{-3} \]

If \( x_1 = (.23113) 10^{-3} \) then \( x_2 = (.23114) 10^{-3} \), hence

\[ x_2 - x_1 = (.00001) 10^{-3} = 10^{-8} \]
5. In a binary machine ($\beta = 2$), the largest number in absolute value is

$$x_{\text{max}} = (.111 \cdots 1) \, 2^U$$

and the smallest is

$$x_{\text{min}} = (.100 \cdots 0) \, 2^L$$

It is easy to see that the set $M$ is not closed under addition or multiplication, since

$$x_{\text{max}} + x_{\text{max}} \notin M$$

and

$$x_{\text{min}} \cdot x_{\text{min}} \notin M$$

Let us see an example in the decimal system ($\beta = 10$) with $t = 7$. Let $x = (.1000000) \, 10^1$ and $y = (.1000000) \, 10^{-6}$. Then

$$x + y = 1 + 0.0000001 = 1.0000001 \notin M$$

When the result of an operation does not correspond to a particular machine number, then the machine will replace it with the closest machine number.

A2. Computer representation of floating point numbers

We now consider only machines using the binary system ($\beta = 2$). According to the ANSI/IEEE (Institute of Electrical Engineers) standard, the length of a machine number is expressed in binary digits\(^1\) and is given by

$$\ell = p + q + 1$$

where $p$ the length of the exponent, $q$ the length of the mantissa, while one memory position is reserved for the sign of the number. The representation of a machine number is as follows:

---

\(^1\) 8bits = 1byte.
The table below shows the values used in the system RS/6000.

<table>
<thead>
<tr>
<th></th>
<th>Single Precision</th>
<th>Double Precision</th>
<th>Quadruple Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sign</strong></td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><strong>Exponent, p</strong></td>
<td>8</td>
<td>11</td>
<td>14</td>
</tr>
<tr>
<td><strong>Mantissa, q</strong></td>
<td>23(+1)</td>
<td>52(+1)</td>
<td>112(+1)</td>
</tr>
<tr>
<td><strong>Length, ℓ</strong></td>
<td>32 bits (4bytes)</td>
<td>64bits (8bytes)</td>
<td>128 bits (16bytes)</td>
</tr>
<tr>
<td><strong>L</strong></td>
<td>-126</td>
<td>-1022</td>
<td>-16382</td>
</tr>
<tr>
<td><strong>U</strong></td>
<td>127</td>
<td>1023</td>
<td>16383</td>
</tr>
<tr>
<td><strong>Decimal q’</strong></td>
<td>7.22</td>
<td>15.95</td>
<td>34.02</td>
</tr>
<tr>
<td><strong>Decimal U’</strong></td>
<td>38.23</td>
<td>307.95</td>
<td>4931.77</td>
</tr>
</tbody>
</table>

There holds:

\[ \beta^U = 10^{U'} \Rightarrow U' = U \log_{10} \beta \]

and

\[ \beta^{q+1} = 10^{q'} \Rightarrow q' = (q + 1) \log_{10} \beta \]

**Single precision**

In single precision, the maximum value for the exponent is

\[ U = (1.11111111)_2 = \frac{2^7 - 1}{2 - 1} = 127 \]

and there holds

\[-126 \leq e \leq 127 \]

**Double precision**

Similarly, in double precision the maximum value for the exponent is

\[ U = (1.11111111111)_2 = \frac{2^{10} - 1}{2 - 1} = 1023 \]

and

\[-1022 \leq e \leq 1023 \]

The minimum and maximum, in absolute value, numbers are

\[ x_{min} = 2^{-1} \cdot 2^L = 2^{L-1} = 2^{-1023} \]

In MATLAB, we have

\[ >> 2^{(-1022)} \]
The commands `realmin` and `realmax` do the same job:

```matlab
>> realmin
ans = 2.2251e-308

>> realmax
ans = 1.7977e+308
```

Get help on the above two commands, as well as on `intmin` and `intmax`.

**A3. Approximation of reals using machine numbers**

We consider the set of machine numbers $M(\beta, t, L, U)$, i.e. the set that includes 0 and the numbers of the form

$$\sigma (\alpha_1\alpha_2 \ldots \alpha_t) \beta^e \text{ where } \alpha_1 \neq 0 \text{ and } L \leq e \leq U$$

(4)

As already mentioned, the maximum in absolute value machine number is

$$x_{\text{max}} = (\beta - 1)(\beta - 1) \ldots (\beta - 1) \beta^U$$

(5)

and the minimum in absolute value number is

$$x_{\text{min}} = (1.00 \ldots 0) \beta^L$$

(6)

Suppose now that a real number

$$x = \sigma (\alpha_1\alpha_2 \ldots \alpha_t\alpha_{t+1} \ldots) \beta^e \text{ with } \alpha_1 \neq 0 \text{ and } e \in \mathbb{Z}$$

(7)

appears in the calculations of a program. The closest machine number to $x \in \mathbb{R}$, is denoted by $fl(x)$.

**Overflow and Underflow**

In case

$$|x| \geq x_{\text{max}}$$

then we say we have an overflow and the computer stops. If
\[ |x| \leq x_{\text{min}} \]
then we say we have an underflow. In this case, the computer sets
\[ fl(x) = 0 \]
and continues with the computations. It is obvious that in both cases, we loose information on the number \( x \). Also,
\[ fl(0) = 0 \]

**Truncation and round-off**

We now consider the case when
\[ x_{\text{min}} \leq |x| \leq x_{\text{max}} \]
If \( x \) equals a machine number, then
\[ fl(x) = x \]
If \( x \) does not equal a machine number, then
\[ x' \leq x \leq x'', \quad x', x'' \in M \]
It is expected that \( fl(x) \) will be either \( x' \) and \( x'' \). In case \( x \) is positive
\[ x' = (\alpha_1 \alpha_2 \cdots \alpha_t) \beta^e \quad \text{and} \quad x'' = (\alpha_1 \alpha_2 \cdots \alpha_t + \beta^{-t}) \beta^e \]
In order to obtain \( fl(x) \) we can either
(a) **truncate (chop)**, or
(b) **round**

In truncation, the digits on the right of \( \alpha_t \) are discarded and
\[ fl(x) = x' = (\alpha_1 \alpha_2 \cdots \alpha_t) \beta^e \]
In rounding, \( fl(x) \) is the closest machine number to \( x \). In order to round a decimal number we
- discard the digits if the first of them is one of 0, 1, 2, 3, 4
- raise the last digit by 1 otherwise

If the number is given in binary form, we
- discard the digits if the first one is 0
- raise the last digit by 1 otherwise

**Examples**

1. We consider decimal (\( \beta = 10 \)) machine numbers with 5 significant digits (\( t = 5 \)). We have
\[ f(l(-24.42695)) \xrightarrow{\text{trunc}} -.24426 \times 10^2 \]

\[ f(l(-24.42695)) \xrightarrow{\text{round}} -.24427 \times 10^2 \]

\[ f(l(0.0003462278)) \xrightarrow{\text{trunc}} .34622 \times 10^{-3} \]

\[ f(l(0.0003462278)) \xrightarrow{\text{round}} .34623 \times 10^{-3} \]

\[ f(l(46729162)) \xrightarrow{\text{trunc}} .46729 \times 10^8 \]

\[ f(l(46729162)) \xrightarrow{\text{round}} .46729 \times 10^8 \]

\[ f(l(-10.99961)) \xrightarrow{\text{trunc}} -.10999 \times 10^2 \]

\[ f(l(-10.99961)) \xrightarrow{\text{round}} -.11000 \times 10^2 \]

2. We consider binary (\( \beta = 2 \)) machine numbers with 5 significant digits (\( t = 5 \)). We have

\[ f(l(101.11101)) \xrightarrow{\text{trunc}} .10111 \times 2^3 \]

\[ f(l(101.11101)) \xrightarrow{\text{round}} .11000 \times 2^3 \]

\[ f(l(11.1101)) \xrightarrow{\text{trunc}} .11110 \times 2^3 \]

\[ f(l(11.1101)) \xrightarrow{\text{round}} .11111 \times 2^3 \]
Exercises

1. Which are the values for `realmin` and `realmax` in MATLAB if single precision is used?

2. Complete the table

<table>
<thead>
<tr>
<th></th>
<th>Half Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sign</strong></td>
<td>1</td>
</tr>
<tr>
<td><strong>Exponent, p</strong></td>
<td>5</td>
</tr>
<tr>
<td><strong>Mantissa, q</strong></td>
<td>10(+1)</td>
</tr>
<tr>
<td><strong>Length, ℓ</strong></td>
<td>16 bits (2bytes)</td>
</tr>
<tr>
<td><strong>L</strong></td>
<td></td>
</tr>
<tr>
<td><strong>U</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Decimal q’</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Decimal U’</strong></td>
<td></td>
</tr>
</tbody>
</table>

3. Use MATLAB to obtain plots for

\[ y = x^7 - 7x^6 + 21x^5 - 35x^4 + 35x^3 - 21x^2 + 7x - 1 \]

as well as

\[ y = (x-1)^7 \]

for \(x = 0.988:0.0001:1.012\) and comment on the results.

4. For (a) – (c) below, answer the following

- What does the program do?
- How many lines of output does it produce?
- What are the last two values of \(x\) being printed?

   (a)
   ```matlab
   x=1;
   while 1+x>1
       x=x/2
       pause(.02)
   end
   ```

   (b)
   ```matlab
   x=1;
   while x+x>x
       x=2*x
   ```
pause(.02)
end

c
x=1;
while x+x>x
    x=x/2
    pause(.02)
end

5. The power series for \( \sin(x) \) is given by
\[
\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots
\]
The MATLAB m-file below uses the above power series to compute the sine of \( x \).

```matlab
function s = powersin(x)

    s = 0;
    t = x;
    n = 1;
    while s+t ~= s
        s = s+t;
        t = -x.^2/((n+1)*(n+2)).*t;
        n = n+2;
    end
```

(a) What causes the while loop to terminate?

(b) For \( x = \pi/2, 11\pi/2, 21\pi/2 \) and \( 31\pi/2 \),

- how accurate is the answer?
- how many terms are needed?
- which is the highest order term used in the power series?

6. Modify the above m-file in order to compute \( \cos(x) \), if
\[
\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \ldots
\]
What results do you get for \( x = 0, \pi/3, \pi/4 \) and \( \pi/2 \)?